28. Practical: A Shoutcast Server

In this chapter you'll develop another important part of what will eventually be a Web-based application for streaming MP3s, namely, the server that implements the Shoutcast protocol for actually streaming MP3s to clients such as iTunes, XMMS,[1] or Winamp.

The Shoutcast Protocol

The Shoutcast protocol was invented by the folks at Nullsoft, the makers of the Winamp MP3 software. It was designed to support Internet audio broadcasting--Shoutcast DJs send audio data from their personal computers to a central Shoutcast server that then turns around and streams it out to any connected listeners.

The server you'll build is actually only half a true Shoutcast server--you'll use the protocol that Shoutcast servers use to stream MP3s to listeners, but your server will be able to serve only songs already stored on the file system of the computer where the server is running.

You need to worry about only two parts of the Shoutcast protocol: the request that a client makes in order to start receiving a stream and the format of the response, including the mechanism by which metadata about what song is currently playing is embedded in the stream.

The initial request from the MP3 client to the Shoutcast server is formatted as a normal HTTP request. In response, the Shoutcast server sends an ICY response that looks like an HTTP response except with the string "ICY"[2] in place of the normal HTTP version string and with different headers. After sending the headers and a blank line, the server streams a potentially endless amount of MP3 data.

The only tricky thing about the Shoutcast protocol is the way metadata about the songs being streamed is embedded in the data sent to the client. The problem facing the Shoutcast designers was to provide a way for the Shoutcast server to communicate new title information to the client each time it started playing a new song so the client could display it in its UI. (Recall from Chapter 25 that the MP3 format doesn't make any provision for encoding metadata.) While one of the design goals of ID3v2 had been to make it better suited for use when streaming MP3s, the Nullsoft folks decided to go their own route and invent a new scheme that's fairly easy to implement on both the client side and the server side. That, of course, was ideal for them since they were also the authors of their own MP3 client.

Their scheme was to simply ignore the structure of MP3 data and embed a chunk of self-delimiting metadata every *n* bytes. The client would then be responsible for stripping out this metadata so it wasn't treated as MP3 data. Since metadata sent to a client that isn't ready for it will cause glitches in the sound, the server is supposed to send metadata only if the client's original request contains a special Icy-Metadata header. And in order for the client to know how often to expect metadata, the server must send back a header Icy-Metaint whose value is the number of bytes of MP3 data that will be sent between each chunk of metadata.

The basic content of the metadata is a string of the form "StreamTitle='*title*';" where *title* is the title of the current song and can't contain single quote marks. This payload is encoded as a length-delimited array of bytes: a single byte is sent indicating how many 16-byte blocks follow, and then that many blocks are sent. They contain the string payload as an ASCII string, with the final block padded out with null bytes as necessary.

Thus, the smallest legal metadata chunk is a single byte, zero, indicating zero subsequent blocks. If the server doesn't need to update the metadata, it can send such an empty chunk, but it must send at least the one byte so the client doesn't throw away actual MP3 data.

Song Sources

Because a Shoutcast server has to keep streaming songs to the client for as long as it's connected, you need to provide your server with a source of songs to draw on. In the Web-based application, each connected client will have a playlist that can be manipulated via the Web interface. But in the interest of avoiding excessive coupling, you should define an interface that the Shoutcast server can use to obtain songs to play. You can write a simple implementation of this interface now and then a more complex one as part of the Web application you'll build in Chapter 29.

The Package

The package for the code you'll develop in this chapter looks like this:

```
(defpackage :com.gigamonkeys.shoutcast
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.id3v2)
  (:export :song
           :file
           :title
           :id3-size
           :find-song-source
           :current-song
           :still-current-p
           :maybe-move-to-next-song
           :*song-source-type*))
```

The idea behind the interface is that the Shoutcast server will find a source of songs based on an ID extracted from the AllegroServe request object. It can then do three things with the song

source it's given.

- Get the current song from the source
- Tell the song source that it's done with the current song
- Ask the source whether the song it was given earlier is still the current song

The last operation is necessary because there may be ways--and will be in Chapter 29--to manipulate the songs source outside the Shoutcast server. You can express the operations the Shoutcast server needs with the following generic functions:

```
(defgeneric current-song (source)
  (:documentation "Return the currently playing song or NIL."))

(defgeneric maybe-move-to-next-song (song source)
  (:documentation
   "If the given song is still the current one update the value
returned by current-song."))

(defgeneric still-current-p (song source)
  (:documentation
   "Return true if the song given is the same as the current-song."))
```

The function `maybe-move-to-next-song` is defined the way it is so a single operation checks whether the song is current and, if it is, moves the song source to the next song. This will be important in the next chapter when you need to implement a song source that can be safely manipulated from two different threads.[3]

To represent the information about a song that the Shoutcast server needs, you can define a class, `song`, with slots to hold the name of the MP3 file, the title to send in the Shoutcast metadata, and the size of the ID3 tag so you can skip it when serving up the file.

```
(defclass song ()
  ((file     :reader file     :initarg :file)
   (title    :reader title    :initarg :title)
   (id3-size :reader id3-size :initarg :id3-size)))
```

The value returned by `current-song` (and thus the first argument to `still-current-p` and `maybe-move-to-next-song`) will be an instance of `song`.

In addition, you need to define a generic function that the server can use to find a song source based on the type of source desired and the request object. Methods will specialize the `type` parameter in order to return different kinds of song source and will pull whatever information they need from the request object to determine which source to return.

```
(defgeneric find-song-source (type request)
  (:documentation "Find the song-source of the given type for the given request."))
```

However, for the purposes of this chapter, you can use a trivial implementation of this interface that always uses the same object, a simple queue of song objects that you can manipulate from the REPL. You can start by defining a class, `simple-song-queue`, and a global variable, `*songs*`, that holds an instance of this class.

```
(defclass simple-song-queue ()
  ((songs :accessor songs :initform (make-array 10 :adjustable t :fill-pointer 0))
   (index :accessor index :initform 0)))

(defparameter *songs* (make-instance 'simple-song-queue))
```

Then you can define a method on `find-song-source` that specializes `type` with an **`EQL`** specializer on the symbol `singleton` and returns the instance stored in `*songs*`.

```
(defmethod find-song-source ((type (eql 'singleton)) request)
  (declare (ignore request))
  *songs*)
```

Now you just need to implement methods on the three generic functions that the Shoutcast server will use.

```
(defmethod current-song ((source simple-song-queue))
  (when (array-in-bounds-p (songs source) (index source))
    (aref (songs source) (index source))))

(defmethod still-current-p (song (source simple-song-queue))
  (eql song (current-song source)))

(defmethod maybe-move-to-next-song (song (source simple-song-queue))
  (when (still-current-p song source)
    (incf (index source))))
```

And for testing purposes you should provide a way to add songs to this queue.

```
(defun add-file-to-songs (file)
  (vector-push-extend (file->song file) (songs *songs*)))

(defun file->song (file)
  (let ((id3 (read-id3 file)))
    (make-instance
     'song
     :file (namestring (truename file))
     :title (format nil "~a by ~a from ~a" (song id3) (artist id3) (album id3))
     :id3-size (size id3))))
```

Implementing Shoutcast

Now you're ready to implement the Shoutcast server. Since the Shoutcast protocol is loosely based on HTTP, you can implement the server as a function within AllegroServe. However, since you need to interact with some of the low-level features of AllegroServe, you can't use the `define-url-function` macro from Chapter 26. Instead, you need to write a regular function that looks like this:

```
(defun shoutcast (request entity)
  (with-http-response
      (request entity :content-type "audio/MP3" :timeout *timeout-seconds*)
    (prepare-icy-response request *metadata-interval*)
    (let ((wants-metadata-p (header-slot-value request :icy-metadata)))
      (with-http-body (request entity)
        (play-songs
         (request-socket request)
         (find-song-source *song-source-type* request)
         (if wants-metadata-p *metadata-interval*))))))
```

Then publish that function under the path `/stream.mp3` like this:[4]

```
(publish :path "/stream.mp3" :function 'shoutcast)
```

In the call to `with-http-response`, in addition to the usual `request` and `entity` arguments, you need to pass `:content-type` and `:timeout` arguments. The `:content-type` argument tells AllegroServe how to set the Content-Type header it sends. And the `:timeout` argument specifies the number of seconds AllegroServe gives the function to generate its response. By default AllegroServe times out each request after five minutes. Because you're going to stream an essentially endless sequence of MP3s, you need much more time. There's no way to tell AllegroServe to *never* time out the request, so you should set it to the value of `*timeout-seconds*`, which you can define to some suitably large value such as the number of seconds in ten years.

```
(defparameter *timeout-seconds* (* 60 60 24 7 52 10))
```

Then, within the body of the `with-http-response` and before the call to `with-http-body` that will cause the response headers to be sent, you need to manipulate the reply that AllegroServe will send. The function `prepare-icy-response` encapsulates the necessary manipulations: changing the protocol string from the default of "HTTP" to "ICY" and adding the Shoutcast-specific headers.[5] You also need, in order to work around a bug in iTunes, to tell AllegroServe not to use *chunked transfer-encoding*.[6] The functions `request-reply-protocol-string`, `request-uri`, and `reply-header-slot-value` are all part of AllegroServe.

```
(defun prepare-icy-response (request metadata-interval)
  (setf (request-reply-protocol-string request) "ICY")
  (loop for (k v) in (reverse
       `((:|icy-metaint| ,(princ-to-string metadata-interval))
         (:|icy-notice1| "<BR>This stream blah blah blah<BR>")
         (:|icy-notice2| "More blah")
         (:|icy-name|    "MyLispShoutcastServer")
         (:|icy-genre|   "Unknown")
         (:|icy-url|     ,(request-uri request))
         (:|icy-pub|     "1")))
     do (setf (reply-header-slot-value request k) v))
  ;; iTunes, despite claiming to speak HTTP/1.1, doesn't understand
  ;; chunked Transfer-encoding. Grrr. So we just turn it off.
  (turn-off-chunked-transfer-encoding request))

(defun turn-off-chunked-transfer-encoding (request)
  (setf (request-reply-strategy request)
        (remove :chunked (request-reply-strategy request))))
```

Within the `with-http-body` of `shoutcast`, you actually stream the MP3 data. The function `play-songs` takes the stream to which it should write the data, the song source, and the metadata interval it should use or **`NIL`** if the client doesn't want metadata. The stream is the socket obtained from the request object, the song source is obtained by calling `find-song-source`, and the metadata interval comes from the global variable `*metadata-interval*`. The type of song source is controlled by the variable

`*song-source-type*`, which for now you can set to `singleton` in order to use the `simple-song-queue` you implemented previously.

```
(defparameter *metadata-interval* (expt 2 12))

(defparameter *song-source-type* 'singleton)
```

The function `play-songs` itself doesn't do much--it loops calling the function `play-current`, which does all the heavy lifting of sending the contents of a single MP3 file, skipping the ID3 tag and embedding ICY metadata. The only wrinkle is that you need to keep track of when to send the metadata.

Since you must send metadata chunks at a fixed intervals, regardless of when you happen to switch from one MP3 file to the next, each time you call `play-current` you need to tell it when the next metadata is due, and when it returns, it must tell you the same thing so you can pass the information to the next call to `play-current`. If `play-current` gets **`NIL`** from the song source, it returns **`NIL`**, which allows the `play-songs` **`LOOP`** to end.

In addition to handling the looping, `play-songs` also provides a **`HANDLER-CASE`** to trap the error that will be signaled when the MP3 client disconnects from the server and one of the writes to the socket, down in `play-current`, fails. Since the **`HANDLER-CASE`** is outside the **`LOOP`**, handling the error will break out of the loop, allowing `play-songs` to return.

```
(defun play-songs (stream song-source metadata-interval)
  (handler-case
      (loop
         for next-metadata = metadata-interval
         then (play-current
               stream
               song-source
               next-metadata
               metadata-interval)
         while next-metadata)
    (error (e) (format *trace-output* "Caught error in play-songs: ~a" e))))
```

Finally, you're ready to implement `play-current`, which actually sends the Shoutcast data. The basic idea is that you get the current song from the song source, open the song's file, and then loop reading data from the file and writing it to the socket until either you reach the end of the file or the current song is no longer the current song.

There are only two complications: One is that you need to make sure you send the metadata at the correct interval. The other is that if the file starts with an ID3 tag, you want to skip it. If you don't worry too much about I/O efficiency, you can implement `play-current` like this:

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song))))
        (with-open-file (mp3 (file song))
          (unless (file-position mp3 (id3-size song))
            (error "Can't skip to position ~d in ~a" (id3-size song) (file song)))
          (loop for byte = (read-byte mp3 nil nil)
             while (and byte (still-current-p song song-source)) do
```

```
                (write-byte byte out)
                (decf next-metadata)
              when (and (zerop next-metadata) metadata-interval) do
                (write-sequence metadata out)
                (setf next-metadata metadata-interval))

          (maybe-move-to-next-song song song-source)))
      next-metadata)))
```

This function gets the current song from the song source and gets a buffer containing the metadata it'll need to send by passing the title to `make-icy-metadata`. Then it opens the file and skips past the ID3 tag using the two-argument form of **`FILE-POSITION`**. Then it commences reading bytes from the file and writing them to the request stream.[7]

It'll break out of the loop either when it reaches the end of the file or when the song source's current song changes out from under it. In the meantime, whenever `next-metadata` gets to zero (if you're supposed to send metadata at all), it writes `metadata` to the stream and resets `next-metadata`. Once it finishes the loop, it checks to see if the song is still the song source's current song; if it is, that means it broke out of the loop because it read the whole file, in which case it tells the song source to move to the next song. Otherwise, it broke out of the loop because someone changed the current song out from under it, and it just returns. In either case, it returns the number of bytes left before the next metadata is due so it can be passed in the next call to `play-current`.[8]

The function `make-icy-metadata`, which takes the title of the current song and generates an array of bytes containing a properly formatted chunk of ICY metadata, is also straightforward.[9]

```
(defun make-icy-metadata (title)
  (let* ((text (format nil "StreamTitle='~a';" (substitute #\Space #\' title)))
         (blocks (ceiling (length text) 16))
         (buffer (make-array (1+ (* blocks 16))
                             :element-type '(unsigned-byte 8)
                             :initial-element 0)))
    (setf (aref buffer 0) blocks)
    (loop
       for char across text
       for i from 1
       do (setf (aref buffer i) (char-code char)))
    buffer))
```

Depending on how your particular Lisp implementation handles its streams, and also how many MP3 clients you want to serve at once, the simple version of `play-current` may or may not be efficient enough.

The potential problem with the simple implementation is that you have to call **`READ-BYTE`** and **`WRITE-BYTE`** for every byte you transfer. It's possible that each call may result in a relatively expensive system call to read or write one byte. And even if Lisp implements its own streams with internal buffering so not every call to **`READ-BYTE`** or **`WRITE-BYTE`** results in a system call, function calls still aren't free. In particular, in implementations that provide user-extensible streams using so-called Gray Streams, **`READ-BYTE`** and **`WRITE-BYTE`** may result in a generic function call under the covers to dispatch on the class of the stream argument. While generic

function dispatch is normally speedy enough that you don't have to worry about it, it's a bit more expensive than a nongeneric function call and thus not something you necessarily want to do several million times in a few minutes if you can avoid it.

A more efficient, if slightly more complex, way to implement `play-current` is to read and write multiple bytes at a time using the functions **`READ-SEQUENCE`** and **`WRITE-SEQUENCE`**. This also gives you a chance to match your file reads with the natural block size of the file system, which will likely give you the best disk throughput. Of course, no matter what buffer size you use, keeping track of when to send the metadata becomes a bit more complicated. A more efficient version of `play-current` that uses **`READ-SEQUENCE`** and **`WRITE-SEQUENCE`** might look like this:

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song)))
            (buffer (make-array size :element-type '(unsigned-byte 8))))
        (with-open-file (mp3 (file song))
          (labels ((write-buffer (start end)
                     (if metadata-interval
                       (write-buffer-with-metadata start end)
                       (write-sequence buffer out :start start :end end)))

                   (write-buffer-with-metadata (start end)
                     (cond
                       ((> next-metadata (- end start))
                        (write-sequence buffer out :start start :end end)
                        (decf next-metadata (- end start)))
                       (t
                        (let ((middle (+ start next-metadata)))
                          (write-sequence buffer out :start start :end middle)
                          (write-sequence metadata out)
                          (setf next-metadata metadata-interval)
                          (write-buffer-with-metadata middle end))))))

            (multiple-value-bind (skip-blocks skip-bytes)
                (floor (id3-size song) (length buffer))

              (unless (file-position mp3 (* skip-blocks (length buffer)))
                (error "Couldn't skip over ~d ~d byte blocks."
                       skip-blocks (length buffer)))

              (loop for end = (read-sequence buffer mp3)
                 for start = skip-bytes then 0
                 do (write-buffer start end)
                 while (and (= end (length buffer))
                            (still-current-p song song-source)))

              (maybe-move-to-next-song song song-source)))))
      next-metadata)))
```

Now you're ready to put all the pieces together. In the next chapter you'll write a Web interface to the Shoutcast server developed in this chapter, using the MP3 database from Chapter 27 as the source of songs.

---

[1]The version of XMMS shipped with Red Hat 8.0 and 9.0 and Fedora no longer knows how to play MP3s because the folks at Red Hat were worried about the licensing issues related to the MP3 codec. To get an XMMS with MP3 support on these versions of

Linux, you can grab the source from `http://www.xmms.org` and build it yourself. Or, see `http://www.fedorafaq.org/#xmms-mp3` for information about other possibilities.

[2]To further confuse matters, there's a different streaming protocol called *Icecast*. There seems to be no connection between the ICY header used by Shoutcast and the Icecast protocol.

[3]Technically, the implementation in this chapter will also be manipulated from two threads--the AllegroServe thread running the Shoutcast server and the REPL thread. But you can live with the race condition for now. I'll discuss how to use locking to make code thread safe in the next chapter.

[4]Another thing you may want to do while working on this code is to evaluate the form `(net.aserve::debug-on :notrap)`. This tells AllegroServe to not trap errors signaled by your code, which will allow you to debug them in the normal Lisp debugger. In SLIME this will pop up a SLIME debugger buffer just like any other error.

[5]Shoutcast headers are usually sent in lowercase, so you need to escape the names of the keyword symbols used to identify them to AllegroServe to keep the Lisp reader from converting them to all uppercase. Thus, you'd write `:|icy-metaint|` rather than `:icy-metaint`. You could also write `:\i\c\y-\m\e\t\a\i\n\t`, but that'd be silly.

[6]The function `turn-off-chunked-transfer-encoding` is a bit of a kludge. There's no way to turn off chunked transfer encoding via AllegroServe's official APIs without specifying a content length because any client that advertises itself as an HTTP/1.1 client, which iTunes does, is supposed to understand it. But this does the trick.

[7]Most MP3-playing software will display the metadata somewhere in the user interface. However, the XMMS program on Linux by default doesn't. To get XMMS to display Shoutcast metadata, press Ctrl+P to see the Preferences pane. Then in the Audio I/O Plugins tab (the leftmost tab in version 1.2.10), select the MPEG Layer 1/2/3 Player (`libmpg123.so`) and hit the Configure button. Then select the Streaming tab on the configuration window, and at the bottom of the tab in the SHOUTCAST/Icecast section, check the "Enable SHOUTCAST/Icecast title streaming" box.

[8]Folks coming to Common Lisp from Scheme might wonder why `play-current` can't just call itself recursively. In Scheme that would work fine since Scheme implementations are required by the Scheme specification to support "an unbounded number of active tail calls." Common Lisp implementations are allowed to have this property, but it isn't required by the language standard. Thus, in Common Lisp the idiomatic way to write loops is with a looping construct, not with recursion.

[9]This function assumes, as has other code you've written, that your Lisp implementation's internal character encoding is ASCII or a superset of ASCII, so you can use **`CHAR-CODE`** to translate Lisp **`CHARACTER`** objects to bytes of ASCII data.